# The Clarity Compendium

A comprehensive knowledge base of the CS:GO source engine
gathered from years of tool development

WIP. Discord: https://discord.gg/ZHfFyqMKSf

Written by the Clarity Dev Team

February 2026

## 1 Introduction

Clarity was a high-end movement assistance and automation tool for Counter-Strike: Global Offensive. The project was discontinued following the release of Counter-Strike 2. It served as an experimental platform for studying and reproducing emergent movement behaviors within CS:GO's source engine.

Built through rigorous reverse-engineering and systematic analysis of the engine's player movement pipeline, the project focused on theorizing novel, high-impact movement features and developing reliable methods of reproducing them. Clarity pioneered many of the automated features now commonplace amongst many TAS and hack-tools. including auto pixelsurf, auto edgebug, auto align, auto ladder bug, and auto fireman, among others.

While numerous tools have since replicated the features Clarity originally introduced, it remains the originator and has always produced the most cutting-edge results in the CS:GO movement space.

The purpose of this document is to share the knowledge gathered throughout Clarity's development. With that context established, we will now examine the technical foundations of the project.

# 2 Jumps

## 2.1 Crouchjump

Let us start with the basics. The crouchjump is one of the most widely used movement mechanics in the game, but many players do not fundamentally understand why or how crouching while on the ground right before a jump actually results in more height than just jumping and then ducking later.

Getting to the bottom of this means tracing through a few different systems: duck state timers, bounding box resizing, in-air origin adjustment, and how the jump impulse is applied depending on your duck state at liftoff, with order of execution in the pipeline mattering the most. This simple mechanic serves as an excellent introduction into the intracacies of the source engine.

### Bounding Boxes

Before looking at any code, it helps to have the hull vector definitions in front of us:

```
VEC_HULL_MIN      = Vector(-16, -16,  0)
VEC_HULL_MAX      = Vector( 16,  16, 72)
VEC_DUCK_HULL_MIN = Vector(-16, -16,  0)
VEC_DUCK_HULL_MAX = Vector( 16,  16, 54)
```

A standing player is 72 units tall; a ducked player is 54 units. The difference is **18 units**. This is important, so keep this in mind.

### FinishDuck() and Origin Adjustment

Now, when a player is ready to transition into the ducked state, the engine calls `FinishDuck()`, which handles the necessary adjustment to the player's origin vector. The relevant snippet from `CCSGameMovement::FinishDuck()` is shown below:

Listing 1: `CCSGameMovement::FinishDuck()` origin adjustment

```
Vector newOrigin = mv->GetAbsOrigin();

if ( player->GetGroundEntity() != NULL || player->GetMoveType() == MOVETYPE_LADDER )
{
    Vector hullMinDelta = VEC_DUCK_HULL_MIN - VEC_HULL_MIN;
    newOrigin -= hullMinDelta;
}
else
{
    Vector hullSizeNormal = VEC_HULL_MAX - VEC_HULL_MIN;
    Vector hullSizeCrouch = VEC_DUCK_HULL_MAX - VEC_DUCK_HULL_MIN;
    Vector viewDelta      = -0.5f * ( hullSizeNormal - hullSizeCrouch );
    newOrigin -= viewDelta;
}

mv->SetAbsOrigin( newOrigin );
```

**On ground.** What happens to our origin when we input a duck command on the ground? Tracing through the function, we get:

```
hullMinDelta = VEC_DUCK_HULL_MIN - VEC_HULL_MIN
          // = (-16, -16, 0) - (-16, -16, 0)
          // = (0, 0, 0)

newOrigin -= (0, 0, 0)    // no change
```

The result is a no-op. When we duck on the ground, our origin does not change. This is intuitive even without looking at the code, but is good to prove.

**In the air.** In-air duck behavior is where things get interesting:

```
hullSizeNormal = VEC_HULL_MAX - VEC_HULL_MIN
               // = (16, 16, 72) - (-16, -16, 0)
               // = (32, 32, 72)

hullSizeCrouch = VEC_DUCK_HULL_MAX - VEC_DUCK_HULL_MIN
               // = (16, 16, 54) - (-16, -16, 0)
               // = (32, 32, 54)

viewDelta = -0.5 * ( (32, 32, 72) - (32, 32, 54) )
          // = -0.5 * (0, 0, 18)
          // = (0, 0, -9)

newOrigin -= (0, 0, -9)    // equivalent to newOrigin.z += 9
```

Ducking in the air shifts the player's origin *upward* by 9 units. That is half the 18-unit hull height delta. The hull is shrinking symmetrically around its center: the bottom rises by 9, the top drops by 9, and the midpoint stays put.

### Why does a duck before our jump give more height?

The bounding box explanation does little in telling us where the extra jump height comes from. In fact, by this logic alone you should get *less* height from a crouchjump. Since we ducked on the ground and are now in the air, haven't we technically forfeited a potential +9 unit gain by ducking in the air instead? The answer is no, but if you thought this, you are following along well.

The key is that `Duck()` is called *before* `CheckJumpButton()` in the movement pipeline, and ducking in CS:GO is not instantaneous. When you press duck, `m_bDucking` is set to `true` immediately, but `FinishDuck()` is not called yet. The full transition is governed by `m_flDuckAmount`, which is driven by `Approach()` at a rate controlled by `m_flDuckSpeed`:

Listing 2: Duck transition logic in `CCSGameMovement::Duck()`

```
bool duckButtonHeld = ( mv->m_nButtons & IN_DUCK ) != 0;

if ( !duckButtonHeld && player->m_flDuckAmount > 0 )
{
    player->m_Local.m_bDucking = true;
}
else if ( duckButtonHeld && player->m_flDuckAmount < 1 )
{
    player->m_Local.m_bDucking = true;
}

// ... later on in the function ...

float duckSpeed = player->m_flDuckSpeed * 0.8f;

player->m_flDuckAmount = Approach( 1.0f, player->m_flDuckAmount,
                                   gpGlobals->frametime * duckSpeed );

if ( player->m_flDuckAmount >= 1.0f || !playerTouchingGround )
{
    FinishDuck();
}
```

`m_bDucking` signals that a duck transition is in progress. It does not mean the duck is complete. The duck only completes once `m_flDuckAmount` reaches 1.0, which `Approach()` increments by

$$\texttt{m_flDuckSpeed} \times 0.8 \times \texttt{frametime}$$

each tick. At the ideal unpenalized `m_flDuckSpeed` of 8.0, the step size per tick is:

$$8.0 \times 0.8 \times \frac{1}{64} = 0.1 \quad \text{(64-tick)} \qquad 8.0 \times 0.8 \times \frac{1}{128} = 0.05 \quad \text{(128-tick)}$$

Reaching 1.0 therefore takes **10 ticks** on 64-tick and **20 ticks** on 128-tick, both of which are equivalent to approximately 156ms. Any duck spam penalty reducing `m_flDuckSpeed` increases these counts proportionally.

Taking a look at the code block once more, one might also notice the second condition to triggering `FinishDuck()` early: `!playerTouchingGround`. If the player is airborne, even before the duck transition is complete, `FinishDuck()` fires immediately. This is the key takeaway, so hold on to it.

We now transition away from `Duck()` and look towards other functions to explain the mechanic. It turns out all of the aforementioned ducking logic is resolved at the top of the tick, but what's downstream? The first relevant function ends up being a function called `StartGravity()`.

`StartGravity()` is a simple function. The essence of this function is that we are subtracting half a tick of gravitational acceleration from our z velocity before any movement occurs.

Listing 3: Simplified `StartGravity()`

```
void CGameMovement::StartGravity( void )
{
float ent_gravity;
if (player->GetGravity())
    ent_gravity = player->GetGravity();
else
    ent_gravity = 1.0;

mv->m_vecVelocity[2] -= (ent_gravity * sv_gravity.GetFloat() * 0.5 * gpGlobals->frametime );
}
```

On a standard 64-tick server:

```
ent_gravity  = 1
sv_gravity   = 800
tickinterval = 1/64

800 * 0.5 * (1/64) = 6.25 // half gravity
```

Leaving this code block and traveling downstream, the next function we care about handles *all* of the jump button logic, aptly named `CheckJumpButton()`. In the context of this mechanic, the function boils down to checking if the player has input a jump command.

Listing 4: Simplified `CCSGameMovement::CheckJumpButton()`

```
// ... player is on ground and pressed jump ...
SetGroundEntity( NULL ); // player is now considered airborne
// ...
float startz = mv->m_vecVelocity[2];
if ( bStandingOnFallingPlayer )
{
    mv->m_vecVelocity[2] = 0.0f;
}
// ducking or standing on another player
else if ( m_pCSPlayer->m_duckUntilOnGround || ( m_pCSPlayer->m_Local.m_bDucking )
       || ( m_pCSPlayer->GetFlags() & FL_DUCKING ) || bStandingOnOtherPlayer )
{
    mv->m_vecVelocity[2] = flGroundFactor * sv_jump_impulse.GetFloat();
}
// everything else
else
{
```

```
    mv->m_vecVelocity[2] += flGroundFactor * sv_jump_impulse.GetFloat();
}
// ...
FinishGravity();
```

Once a jump command is confirmed, the function immediately calls `SetGroundEntity( NULL )`, which is the game's effective definition of being airborne in the same tick the jump command was received.

But the next branch of logic is where the magic happens. Recall that `StartGravity()` has already been called upstream, so at this point in the pipeline our z velocity sits at $-6.25$ on 64-tick. The two relevant branches behave very differently in light of this:

```
mv->m_vecVelocity[2]  = sv_jump_impulse    // overwrites the negative value entirely
mv->m_vecVelocity[2] += sv_jump_impulse    // adds to the negative value
```

The `=` branch discards the $-6.25$ entirely, yielding a clean `sv_jump_impulse` as the initial upward velocity. The `+=` branch adds the impulse on top of it, yielding `sv_jump_impulse` $- 6.25$. The difference between the two is exactly the amount `StartGravity()` subtracted, which is half a tick of gravity. Crucially, the overwrite branch is the one triggered when `m_bDucking` is `true`, meaning that we enjoy a slightly higher initial velocity than a normal jump.

To summarize, let us trace through a crouchjump tick by tick, where tick 0 is defined as the tick on which the player inputs jump. In this scenario, our imaginary player holds the duck key for approximately 5 ticks before jumping.

| **Tick** | **Event** |
|---|---|
| $-5$ | Player presses duck. `Duck()` sets `m_bDucking = true` and begins advancing `m_flDuckAmount` toward 1.0. Bounding box is unchanged, no origin shift occurs — the player is not yet ducked. |
| $-4$ to $-1$ | `Duck()` increments `m_flDuckAmount` each tick. `playerTouchingGround` is still true so `FinishDuck()` is not called. The player remains mid-transition. |
| 0 | Player presses jump. `Duck()` runs first — `m_bDucking` is still `true`, transition still incomplete. `StartGravity()` brings z velocity to $-6.25$. `CheckJumpButton()` reads `m_bDucking = true` and overwrites z velocity with a clean `sv_jump_impulse` rather than adding to the existing $-6.25$. `SetGroundEntity( NULL )` marks the player as airborne. |
| $+1$ | `Duck()` runs. `playerTouchingGround` is now `false`. The `!playerTouchingGround` condition fires `FinishDuck()` immediately. Since the player is in the air, origin shifts up by 9 units and the bounding box shrinks to 54 units. The crouchjump is complete. |

## 2.2 Longjump

The crouchjump is a commonly used mechanic by all players because it can be performed without alias binds or TAS/hack-tools. The 156ms window to perform a jump command after ducking is a forgiving one and easy to manually reproduce. The critical flaw of a crouchjump is that we begin slowing down immediately after pressing the duck button. `HandleDuckingSpeedCrop()` applies a speed modifier to our horizontal speed whenever `IN_DUCK` is held or `m_bDucking` is true, meaning every grounded tick we spend in the duck transition costs us horizontal velocity:

Listing 5: `CCSGameMovement::HandleDuckingSpeedCrop()`

```
if ( ( mv->m_nButtons & IN_DUCK ) || ( player->m_Local.m_bDucking ) || ( player->GetFlags() & FL_DUCKING
    ) )
{
    float duckSpeedModifier = GetDuckSpeedModifier( duckFraction );
    mv->m_flForwardMove *= duckSpeedModifier;
    mv->m_flSideMove    *= duckSpeedModifier;
    mv->m_flUpMove      *= duckSpeedModifier;
    mv->m_flMaxSpeed    *= duckSpeedModifier;
    m_iSpeedCropped     |= SPEED_CROPPED_DUCK;
}
```

So while a crouchjump is easily reproducible, it is only really useful for jumping to tall surfaces in close proximity. It does little to assist in jumps where distance and horizontal speed are crucial. But what if we can get the jump velocity of a crouchjump while never getting our speed cropped? This is actually possible and is the essence of a **longjump**. The execution follows the exact same principles as a crouchjump, only we are inputting jump and duck **in the same tick** rather than ducking a couple ticks before the jump.

The reason this never slows us down is simple and comes down to execution order in the movement pipeline. `HandleDuckingSpeedCrop()` is the first thing called in `Duck()`, before the game processes any ducking logic. On the tick we send both inputs simultaneously, `m_bDucking` is still `false` when the crop check runs so no penalty is applied. `CheckJumpButton()` fires later in the same tick, sets `m_bDucking = true`, and classifies us as airborne. On every tick after that we are in the air, so the grounded crop condition **never gets the chance to execute**.

| Tick | Event |
|---|---|
| 0 | Player sends `+duck` and `+jump` simultaneously. `HandleDuckingSpeedCrop()` runs first. `m_bDucking` is still `false` so no penalty is applied. `Duck()` sets `m_bDucking = true`. `StartGravity()` brings z velocity to $-6.25$. `CheckJumpButton()` reads `m_bDucking = true` and overwrites z velocity with a clean `sv_jump_impulse`. Player is airborne. |
| +1 | `Duck()` runs. `playerTouchingGround` is `false`. `FinishDuck()` fires immediately. Origin shifts up 9 units, bounding box shrinks to 54 units. |

The longjump produces identical height to a crouchjump. Same impulse overwrite, same origin shift, same hull reduction, but with full horizontal velocity preserved at liftoff.

## 2.3 Minijump

There is a critical consideration for crouchjumps and longjumps that was not disclosed as a build up to this section. There is a small nuance to both mechanics, that is, **you must still be holding duck on the tick after your jump**. If a player executes a crouchjump or longjump but releases duck on the very next tick, the jump becomes bugged and ends up with *less* height than a normal standing jump. This is known as the **minijump** and it is not so much harder to understand now that we have a solid understanding of the primary ducking and jumping mechanisms.

Recall that on tick 0, `FinishDuck()` never fires while the player is grounded when `Duck()` runs. The player has `m_bDucking = true`, the hull is still 72 units, and the origin is untouched. On tick +1, if the player releases duck, the unduck path in `Duck()` fires `FinishUnDuck()` immediately via `!playerTouchingGround`:

Listing 6: `CCSGameMovement::FinishUnDuck()` in-air branch

```
Vector hullSizeNormal = VEC_HULL_MAX - VEC_HULL_MIN;
Vector hullSizeCrouch = VEC_DUCK_HULL_MAX - VEC_DUCK_HULL_MIN;
Vector viewDelta      = -0.5f * ( hullSizeNormal - hullSizeCrouch );
newOrigin += viewDelta;    // origin.z -= 9
```

This is the exact same `viewDelta` as `FinishDuck()`, but the operator is `+=` instead of `-=`. Where `FinishDuck()` shifts the origin *up* by 9 units, `FinishUnDuck()` shifts it *down* by 9 units. The critical detail is that `FinishUnDuck()` has no guard checking whether `FinishDuck()` was ever called. It blindly applies the in-air origin correction as if transitioning from a 54-unit hull back to 72, but the hull was already 72 units since `FinishDuck()` never fired. The player takes a free $-9$ unit origin hit with no corresponding hull change.

What if the player just re-ducks to recover? `FinishDuck()` would then fire in air on the following tick and apply +9, which exactly cancels the $-9$. The origin is restored to where it was right after the jump. But notice what that means: the +9 and $-9$ cancel to zero, leaving no net origin shift above the jump start position. This produces the same arc as a standing full-impulse jump, not a crouchjump. The small velocity advantage from the `=` overwrite is only worth a fraction of a unit of apex height and does not come close to compensating for the missing net +9 shift. To summarize:

| **Jump type** | **Net origin shift** |
|---|---|
| Crouchjump / Longjump | +9 units |
| Standing jump | 0 units |
| Minijump with re-duck | 0 units ($-9$ and +9 cancel) |
| Minijump, no re-duck | $-9$ units |

A minijump without recovery undershoots a standing jump by 9 units. A minijump with re-duck produces the same height as a standing jump. Neither comes close to a crouchjump or longjump.

## 3 placeholder

### 3.1 Bhop

### 3.2 Jumpbug

### 3.3 Stamina hop

### 3.4 Infinite Duck

## 4 Bugs

### 4.1 Edgebug

### 4.2 Edgeskip

### 4.3 Rampbug

### 4.4 Non-AA Wall stucks / wall bug

### 4.5 AA-Wall stucks (coordinate wall bugs)

## 5 Pixels

### 5.1 Alignment

### 5.2 Pixelsurf

### 5.3 Upwards pixelsurf

### 5.4 Pixeljump

### 5.5 Pixelbounce

### 5.6 Pixelwalks

## 6 Ladders

### 6.1 Fast laddering

### 6.2 Ladder glide

### 6.3 Ladder bug

### 6.4 Fireman

### 6.5 Ladder super jumps

### 6.6 Super ladder bug

## 7 Players

### 7.1 Runboosts

### 7.2 Headbounces

## 8 Strafing

### 8.1 Autostrafe / theoretical perfect strafes

### 8.2 Speed cut

### 8.3 Wall strafes